

Hibernate / JPA Overview



Topics

- What is Hibernate?
- Benefits of Hibernate
- What is JPA?
- Benefits of JPA
- Code Snippets

What is Hibernate?

- A framework for persisting / saving Java objects in a database
- www.hibernate.org/orb



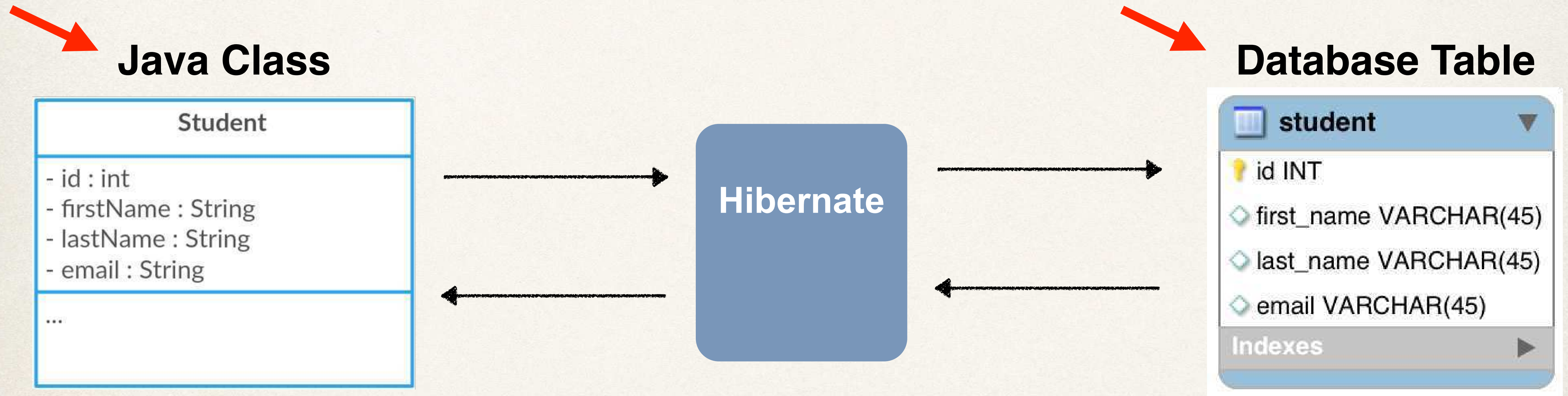
Benefits of Hibernate

- Hibernate handles all of the low-level SQL
- Minimizes the amount of JDBC code you have to develop
- Hibernate provides the Object-to-Relational Mapping (ORM)



Object-To-Relational Mapping (ORM)

- The developer defines mapping between Java class and database table

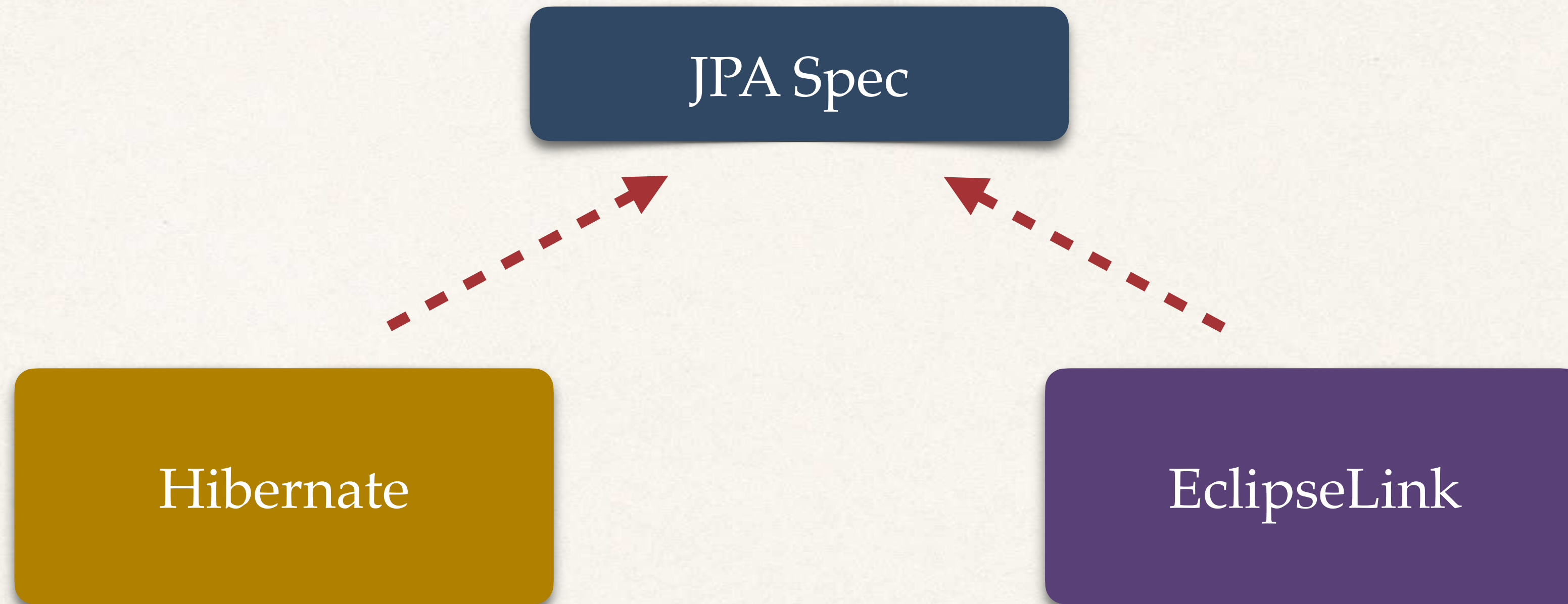


What is JPA?

- Jakarta Persistence API (JPA) ... *previously known as Java Persistence API*
- Standard API for Object-to-Relational-Mapping (ORM)
- Only a specification
- Defines a set of interfaces
- Requires an implementation to be usable

www.luv2code.com/jpa-spec

JPA - Vendor Implementations

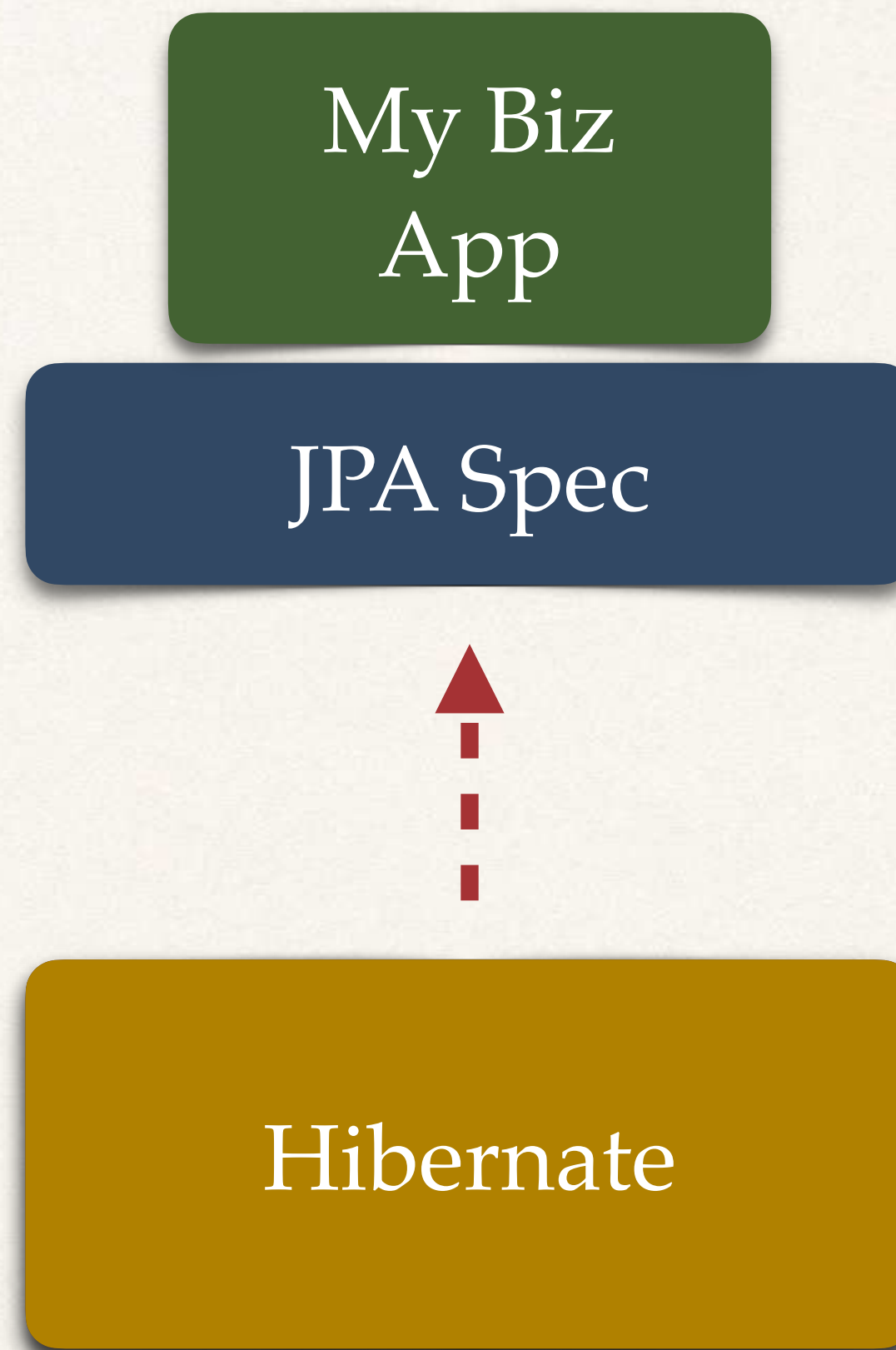


www.luv2code.com/jpa-vendors

What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
 - For example, if Vendor ABC stops supporting their product
 - You could switch to Vendor XYZ without vendor lock in

JPA - Vendor Implementations



Saving a Java Object with JPA

```
// create Java object
Student theStudent = new Student("Paul", "Doe", "paul@luv2code.com");

// save it to database
entityManager.persist(theStudent);
```

Special JPA helper object

The data will be stored in the database

SQL insert

Retrieving a Java Object with JPA

```
// create Java object  
Student theStudent = new Student("Paul", "Doe", "paul@luv2code.com");  
  
// save it to database  
entityManager.persist(theStudent);  
  
// now retrieve from database using the primary key  
int theId = 1;  
Student myStudent = entityManager.find(Student.class, theId);
```

Query the database for given id

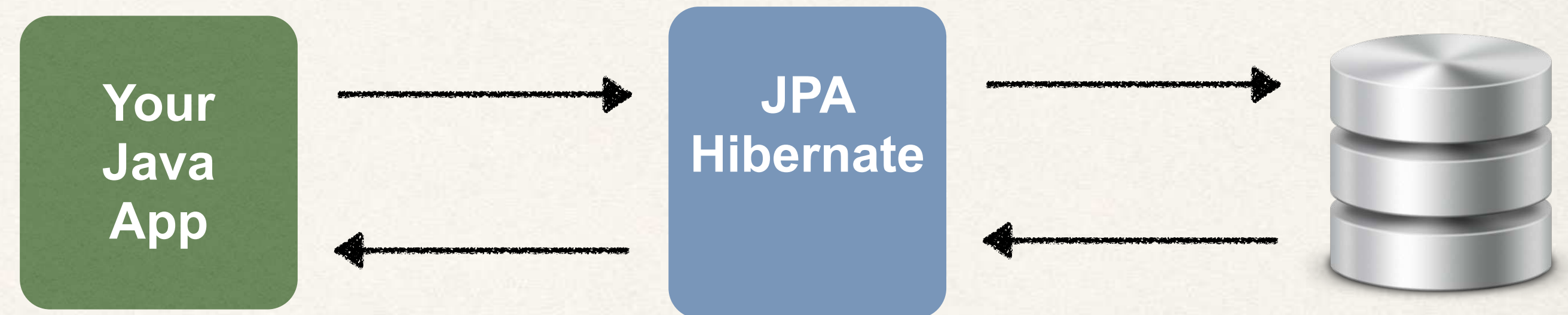
Querying for Java Objects

```
TypedQuery<Student> theQuery = entityManager.createQuery("from Student", Student.class);  
List<Student> students= theQuery.getResultList();
```

Returns a list of Student objects
from the database

JPA/Hibernate CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects



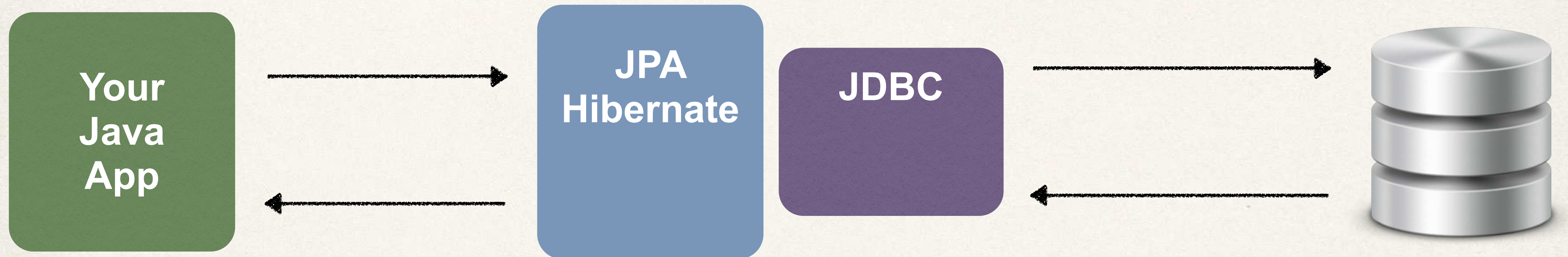
Hibernate / JPA and JDBC



How does Hibernate / JPA
relate to JDBC?

Hibernate / JPA and JDBC

- Hibernate / JPA uses JDBC for all database communications



MySQL Database



MySQL Database

- In this course, we will use the MySQL Database
- MySQL includes two components
 - MySQL Database Server
 - MySQL Workbench

MySQL Database Server

- The MySQL Database Server is the main engine of the database
- Stores data for the database
- Supports CRUD features on the data

MySQL Workbench

- MySQL Workbench is a client GUI for interacting with the database
- Create database schemas and tables
- Execute SQL queries to retrieve data
- Perform insert, updates and deletes on data
- Handle administrative functions such as creating users
- Others ...

Install the MySQL software

- Step 1: Install MySQL Database Server
 - <https://dev.mysql.com/downloads/mysql/>
- Step 2: Install MySQL Workbench
 - <https://dev.mysql.com/downloads/workbench/>

**Please install the
MySQL software now**

Setup Database Table



Two Database Scripts

1. Folder: 00-starter-sql-scripts

- 01-create-user.sql
- 02-student-tracker.sql

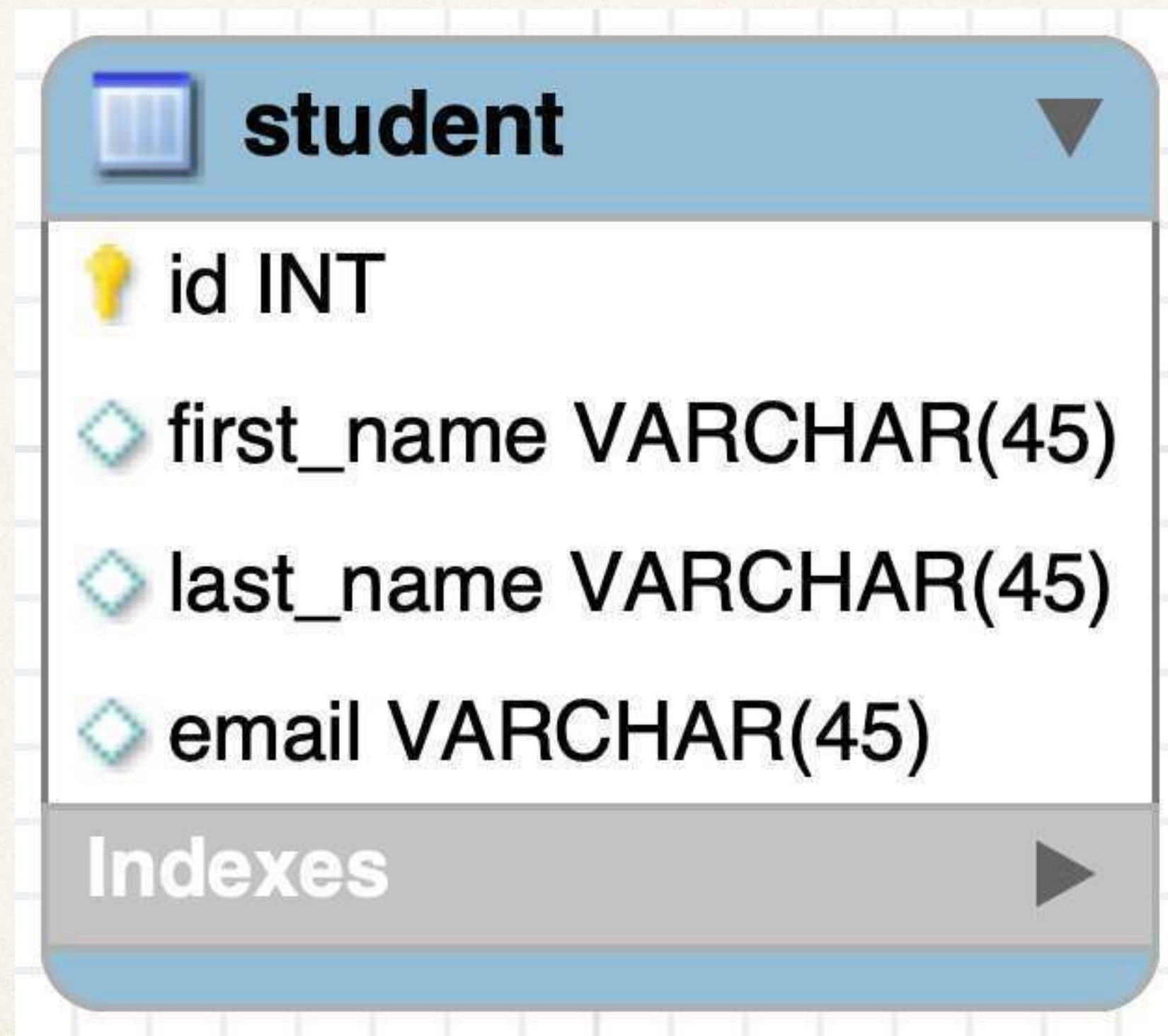
About: 01-create-user.sql

1. Create a new MySQL user for our application

- user id: **springstudent**
- password: **springstudent**

About: 02-student-tracker.sql

1. Create a new database table: **student**



A screenshot of a database table definition for a table named 'student'. The table is shown in a light blue header bar with a table icon and a dropdown arrow. Below the header, the table's columns are listed: 'id' is an integer (INT) and is marked as the primary key with a yellow key icon; 'first_name', 'last_name', and 'email' are all variable character strings (VARCHAR) with a length of 45, each marked with a light blue diamond icon. At the bottom of the table definition, there is a grey bar labeled 'Indexes' with a right-pointing arrow.

student	
id	INT
first_name	VARCHAR(45)
last_name	VARCHAR(45)
email	VARCHAR(45)
Indexes	

Setting Up Spring Boot Project



Automatic Data Source Configuration

- In Spring Boot, Hibernate is the default implementation of JPA
- **EntityManager** is main component for creating queries etc ...
- **EntityManager** is from Jakarta Persistence API (JPA)

Automatic Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
 - **DataSource, EntityManager, ...**
- You can then inject these into your app, for example your DAO

Setting up Project with Spring Initializr

- At Spring Initializr website, start.spring.io
- Add dependencies
 - MySQL Driver: **mysql-connector-j**
 - Spring Data JPA: **spring-boot-starter-data-jpa**

Spring Boot - Auto configuration

- Spring Boot will automatically configure your data source for you
- Based on entries from Maven pom file
 - JDBC Driver: **mysql-connector-j**
 - Spring Data (ORM): **spring-boot-starter-data-jpa**
- DB connection info from **application.properties**

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/student_tracker  
spring.datasource.username=springstudent  
spring.datasource.password=springstudent
```

**No need to give JDBC driver class name
Spring Boot will automatically detect it based on URL**

Creating Spring Boot - Command Line App

- We will create a Spring Boot - Command Line App
- This will allow us to focus on Hibernate / JPA
- Later in the course, we will apply this to a CRUD REST API

Creating Spring Boot - Command Line App

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean
```

```
@SpringBootApplication
```

```
public class CruddemoApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(CruddemoApplication.class, args);
    }
```

```
    @Bean
```

```
    public CommandLineRunner commandLineRunner(String[] args) {
```

```
        return runner -> {
            System.out.println("Hello world");
        };
    }
```

```
}
```

Executed after the
Spring Beans have been loaded

Lambda
expression

Add our
custom code

JPA Development Process



JPA Dev Process - To Do List

1. Annotate Java Class
2. Develop Java Code to perform database operations

Let's just say “JPA”

- As mentioned, Hibernate is the default JPA implementation in Spring Boot
- Going forward in this course, I will simply use the term: **JPA**
- Instead of saying “JPA Hibernate”
- We know that by default, Hibernate is used behind the scenes

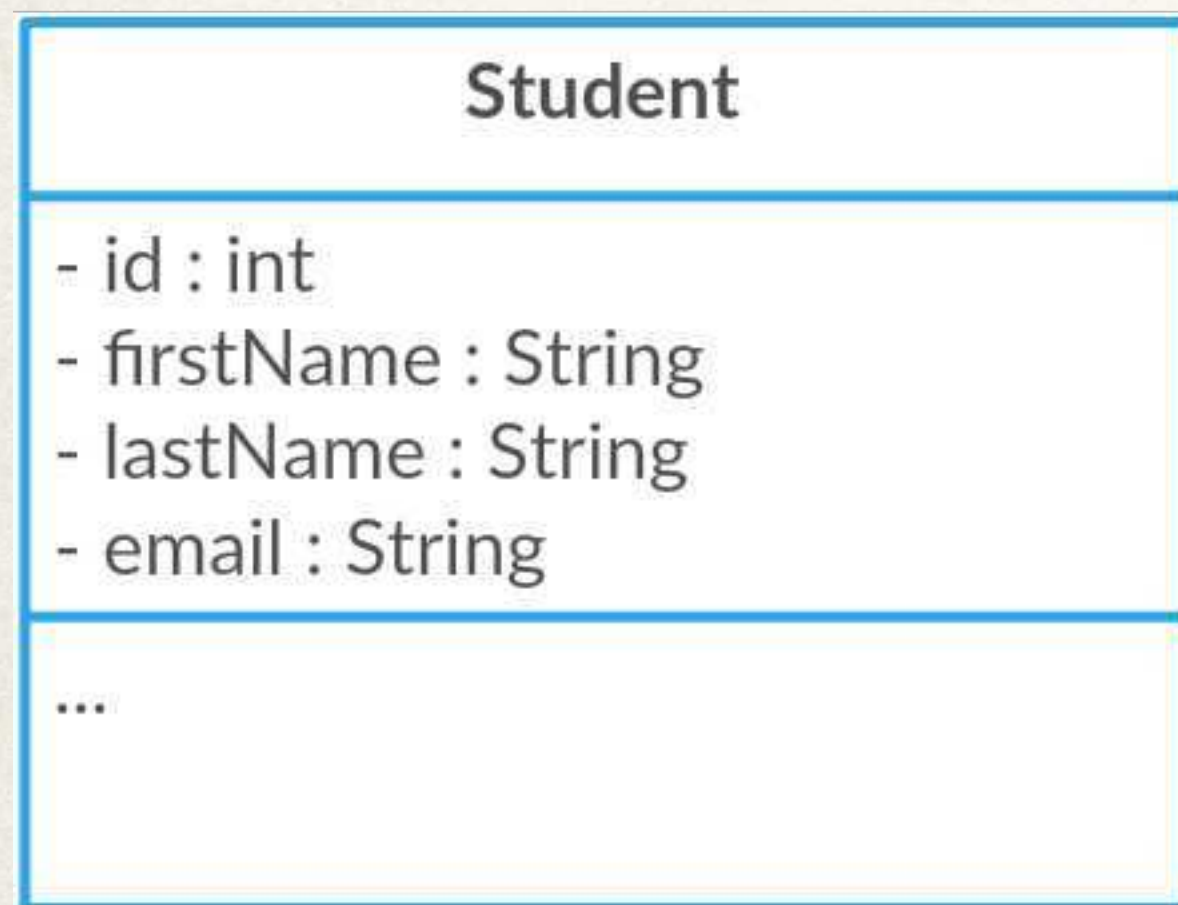
Terminology

Entity Class

Java class that is mapped to a database table

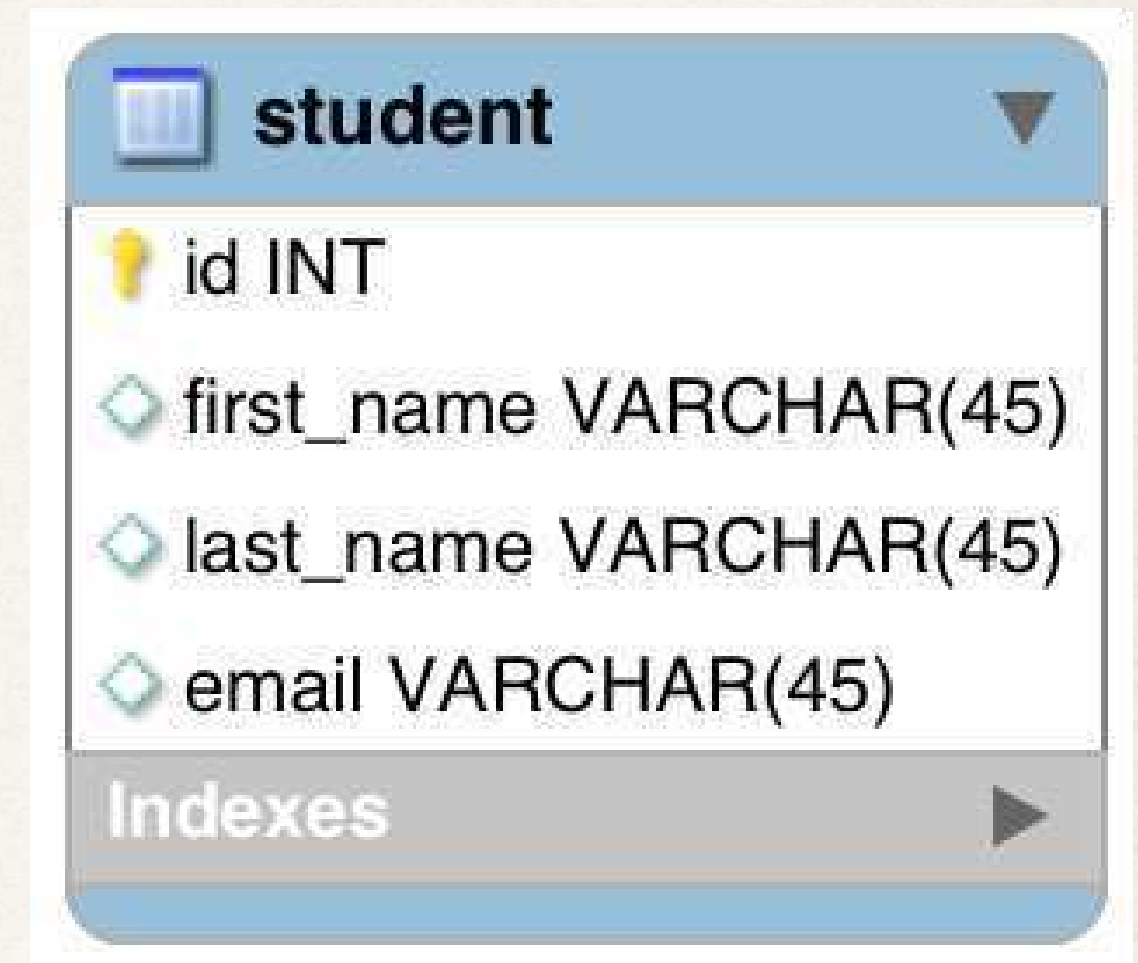
Object-to-Relational Mapping (ORM)

Java Class



JPA

Database Table



Entity Class

- At a minimum, the Entity class
 - Must be annotated with @Entity
 - Must have a public or protected no-argument constructor
 - The class can have other constructors

Constructors in Java - Refresher

- Remember about constructors in Java
- If you don't declare any constructors
 - Java will provide a no-argument constructor for free
- If you declare constructors with arguments
 - then you do NOT get a no-argument constructor for free
 - In this case, you have to explicitly declare a no-argument constructor

Java Annotations

- Step 1: Map class to database table
- Step 2: Map fields to database columns

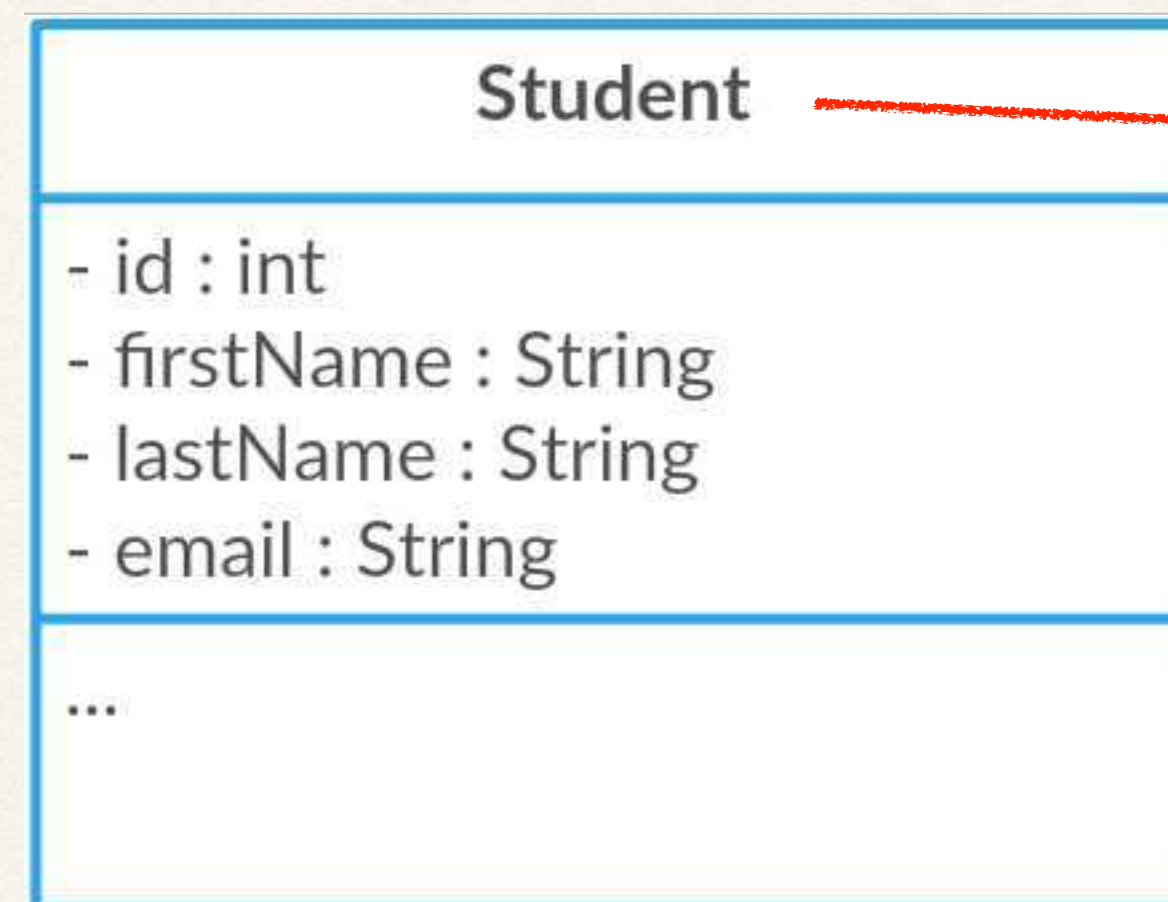
Step 1: Map class to database table

```
@Entity
@Table(name="student")
public class Student {

    ...

}
```

Java Class



Database Table



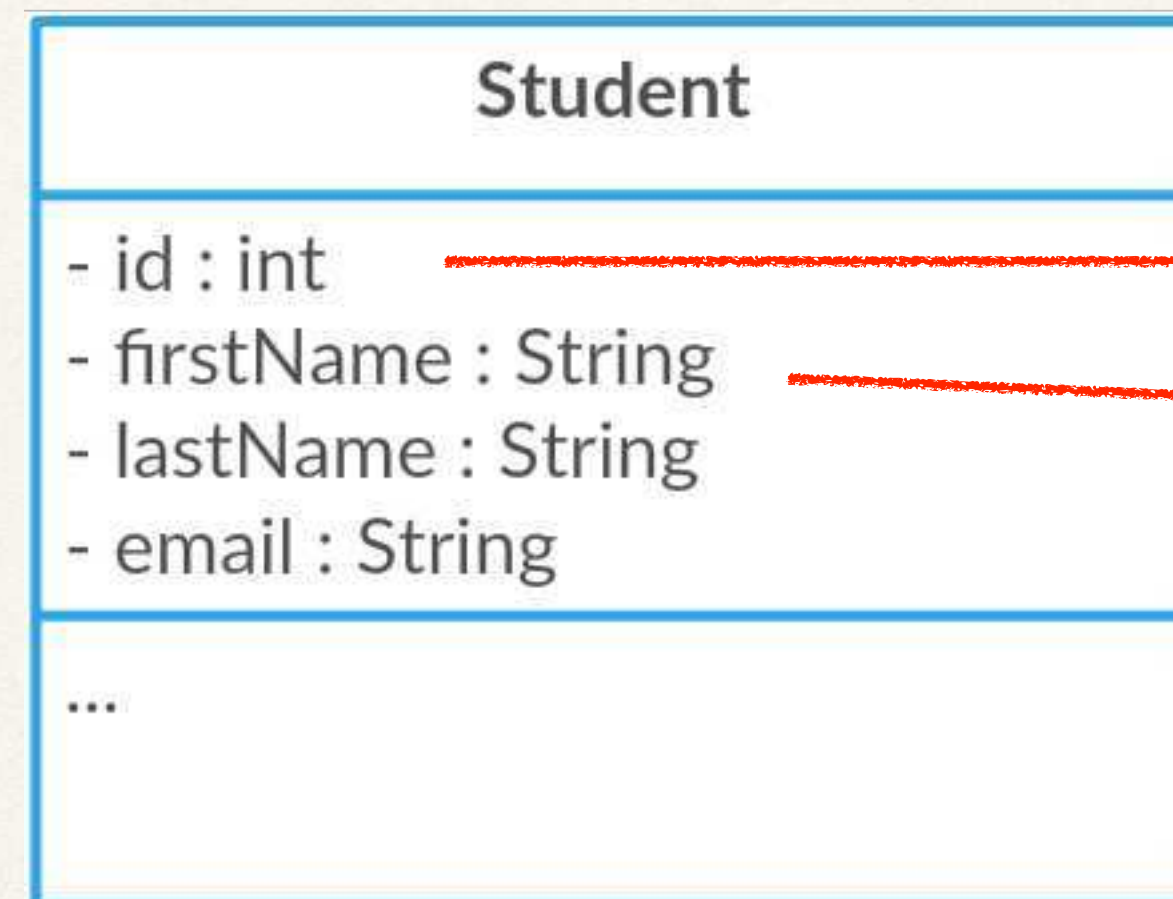
Step 2: Map fields to database columns

```
@Entity
@Table(name="student")
public class Student {

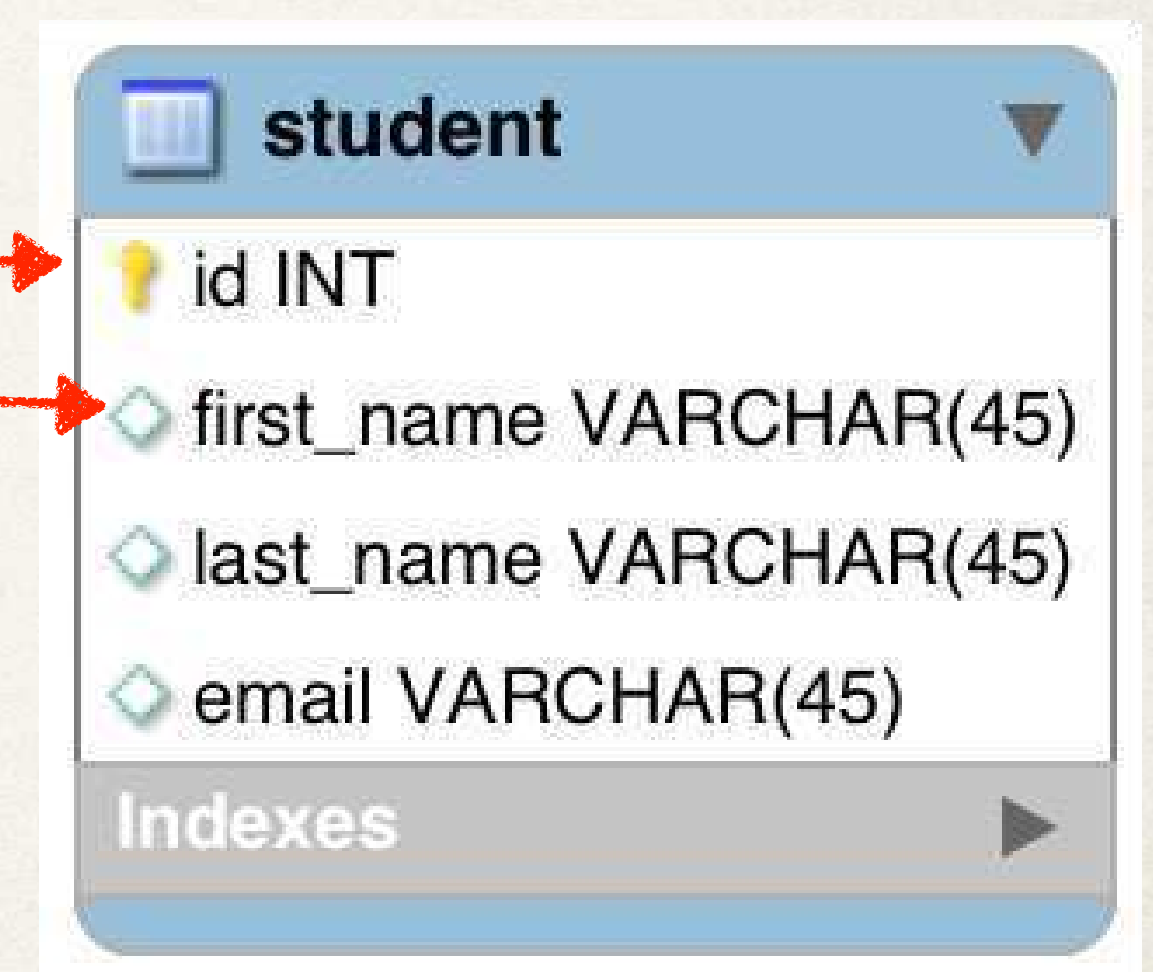
    @Id
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;
    ...
}
```

Java Class



Database Table



@Column - Optional

- Actually, the use of @Column is optional
- If not specified, the column name is the same name as Java field
- In general, I don't recommend this approach
 - If you refactor the Java code, then it will not match existing database columns
 - This is a breaking change and you will need to update database column
- Same applies to @Table, database table name is same as the class

Terminology

Primary Key

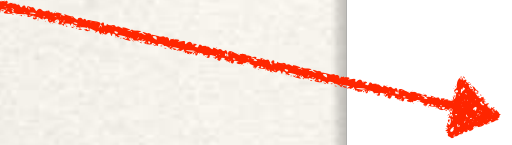
Uniquely identifies each row in a table

Must be a unique value

Cannot contain NULL values

MySQL - Auto Increment

```
CREATE TABLE student (
```



```
    id int NOT NULL AUTO_INCREMENT,  
    first_name varchar(45) DEFAULT NULL,  
    last_name varchar(45) DEFAULT NULL,  
    email varchar(45) DEFAULT NULL,  
    PRIMARY KEY (id)
```

```
)
```


JPA Identity - Primary Key

```
@Entity
```

```
@Table(name="student")
```

```
public class Student {
```

```
    @Id
```

```
    @GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
    @Column(name="id")
```

```
    private int id;
```

```
    ...
```

```
}
```


ID Generation Strategies

Name	Description
GenerationType.AUTO	Pick an appropriate strategy for the particular database
GenerationType.IDENTITY	Assign primary keys using database identity column
GenerationType.SEQUENCE	Assign primary keys using a database sequence
GenerationType.TABLE	Assign primary keys using an underlying database table to ensure uniqueness
GenerationType.UUID	Assign primary keys using a globally unique identifier (UUID) to ensure uniqueness

Bonus Bonus

- You can define your own CUSTOM generation strategy :-)
- Create implementation of **`org.hibernate.id.IdentifierGenerator`**
- Override the method: **`public Serializable generate(...)`**

Save a Java Object



Sample App Features

→ Create a new Student

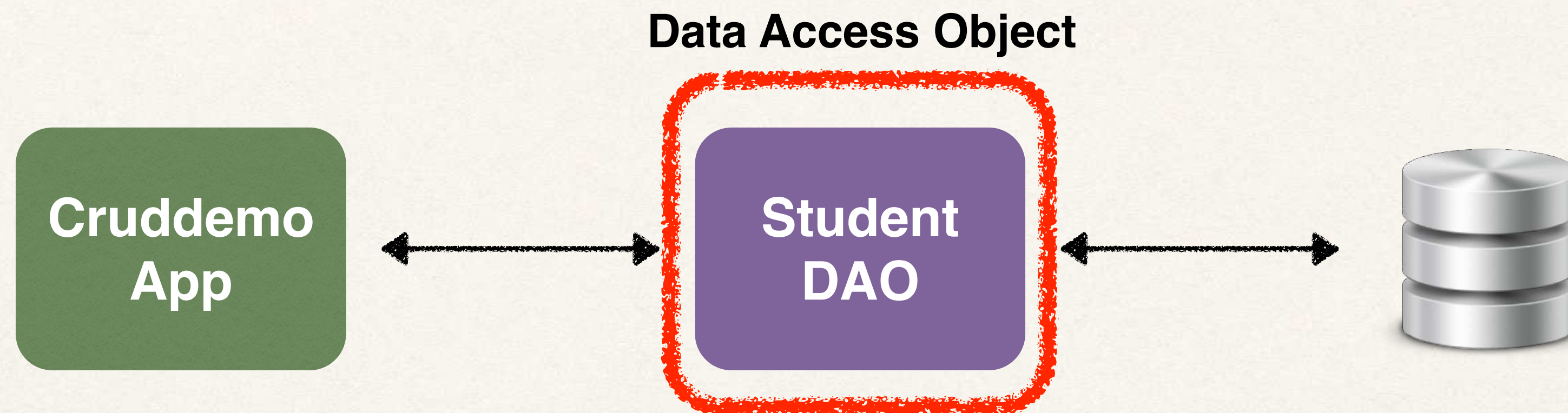
- Read a Student
- Update a Student
- Delete a Student



CRUD

Student Data Access Object

- Responsible for interfacing with the database
- This is a common design pattern: Data Access Object (DAO)



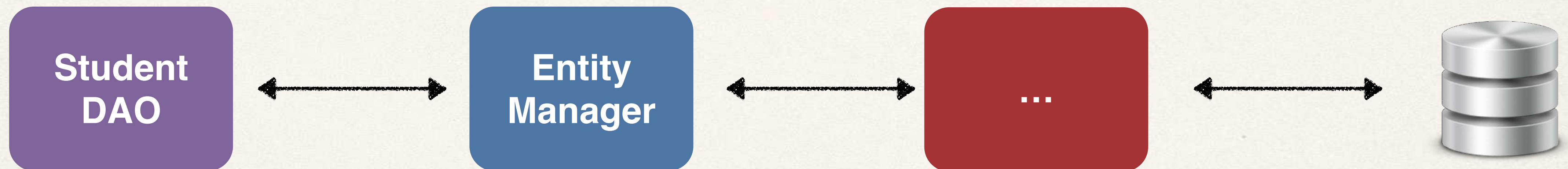
Student Data Access Object

Methods
save(...)
findById(...)
findAll()
findByLastName(...)
update(...)
delete(...)
deleteAll()

Student Data Access Object

- ❖ Our DAO needs a JPA Entity Manager
- ❖ JPA Entity Manager is the main component for saving / retrieving entities

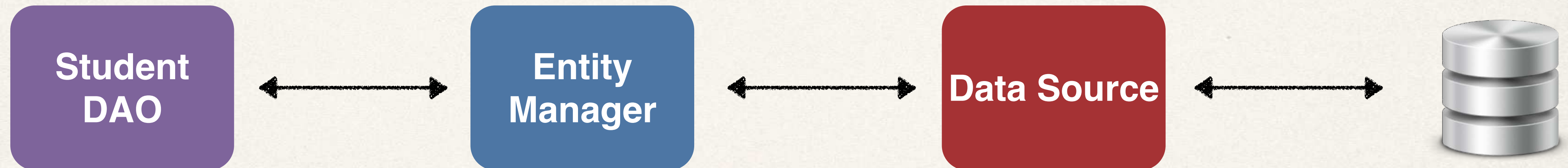
Data Access Object



JPA Entity Manager

- Our JPA Entity Manager needs a Data Source
- The Data Source defines database connection info
- JPA Entity Manager and Data Source are automatically created by Spring Boot
 - Based on the file: application.properties (JDBC URL, user id, password, etc ...)
- We can autowire / inject the JPA Entity Manager into our Student DAO

Data Access Object



What about JpaRepository???

- Spring Data JPA has a JpaRepository interface
- This provides JPA database access with minimal coding

JpaRepository

Why we have not simply used JpaRepository or CRUD Repository here instead of using EntityManager and all?

VF

EntityManager vs JpaRepository

I've seen a bunch of Spring tutorials in Youtube and they usually use JpaRepository instead of EntityManager

Is there a specific context that I should choose one between them or it's just a personal choice??

Which One EntityManager or JpaRepository???

IB

EntityManager and JpaRepository

what is the difference between EntityManager and JpaRepository , why in Section3 you are working with entityManager and are you going to use JpaRepository in other sections??

AG

Why do we use entity manager when we can use JpaRepository or CRUDRepository?

I have some experience in springboot before. So when creating a repository interface , we extend it to JpaRepository or CRUDRepository , which already has inbuilt functions for findAll() or findById() etc.. Im just wondering what the difference is between entity manager and JpaRepository or CRUDRepository? does using entity manager make the code run better?

Answer

- Yes, we will use JpaRepository in this course
- We will cover it later in the course
- In this course, I want to show you various techniques for using JPA
- Knowing BOTH EntityManager and JpaRepository will help you on future projects
- Don't worry ... we'll cover both :-)

In Simple Terms

- If you need **low-level control and flexibility**, use **EntityManager**
- If you want **high-level of abstraction**, use **JpaRepository**

Use Case

Entity Manager

- Need low-level control over the database operations and want to write custom queries
- Provides low-level access to JPA and work directly with JPA entities
- Complex queries that required advanced features such as native SQL queries or stored procedure calls
- When you have custom requirements that are not easily handled by higher-level abstractions

JpaRepository

- Provides commonly used CRUD operations out of the box, reducing the amount of code you need to write
- Additional features such as pagination, sorting
- Generate queries based on method names
- Can also create custom queries using @Query

My Recommendation

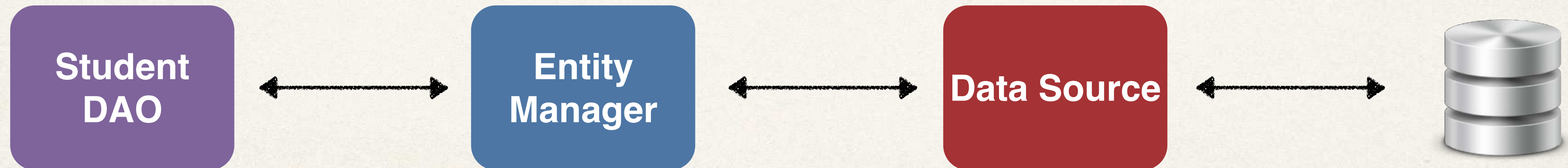
- Choice depends on the application requirements and developer preference
- You can also use both in the same project
- For learning purposes, start with EntityManager then learn JpaRepository
- This will help you understand the low-level coding behind the scenes
- Knowing BOTH EntityManager and JpaRepository will help you on future projects

Student DAO

- Step 1: Define DAO interface
- Step 2: Define DAO implementation
 - Inject the entity manager
- Step 3: Update main app

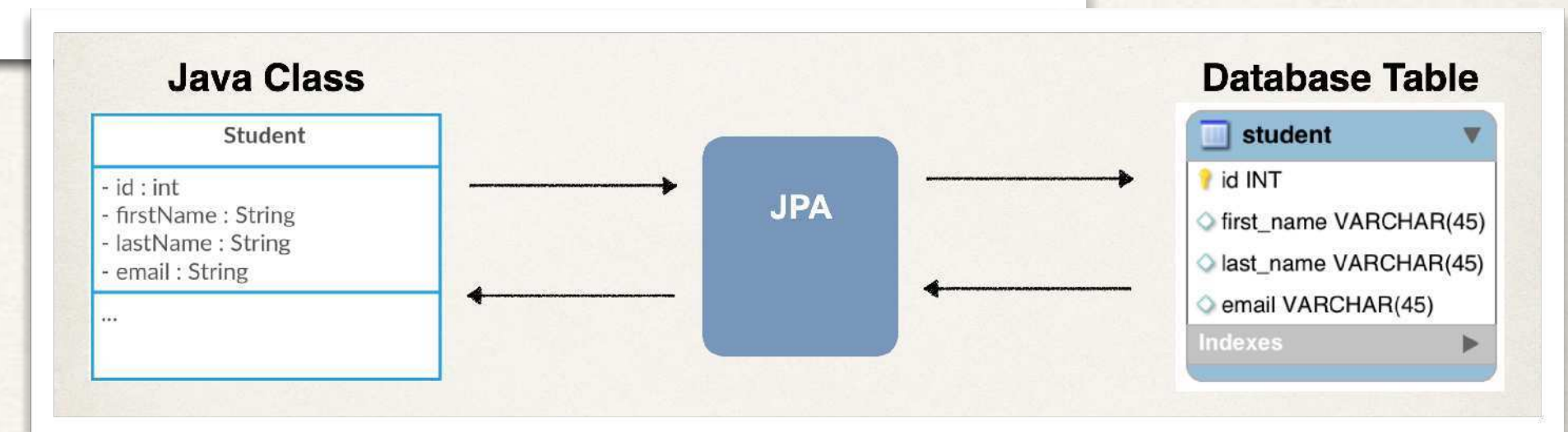
Step-By-Step

Data Access Object



Step 1: Define DAO interface

```
import com.luv2code.cruddemo.entity.Student;  
  
public interface StudentDAO {  
    → void save(Student theStudent);  
}
```



Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

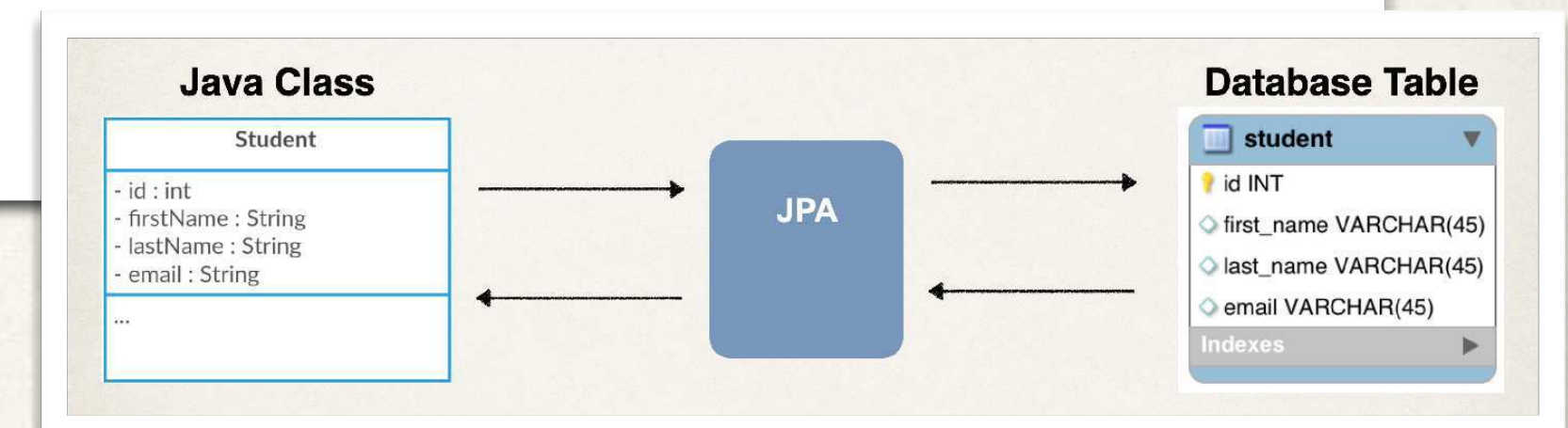
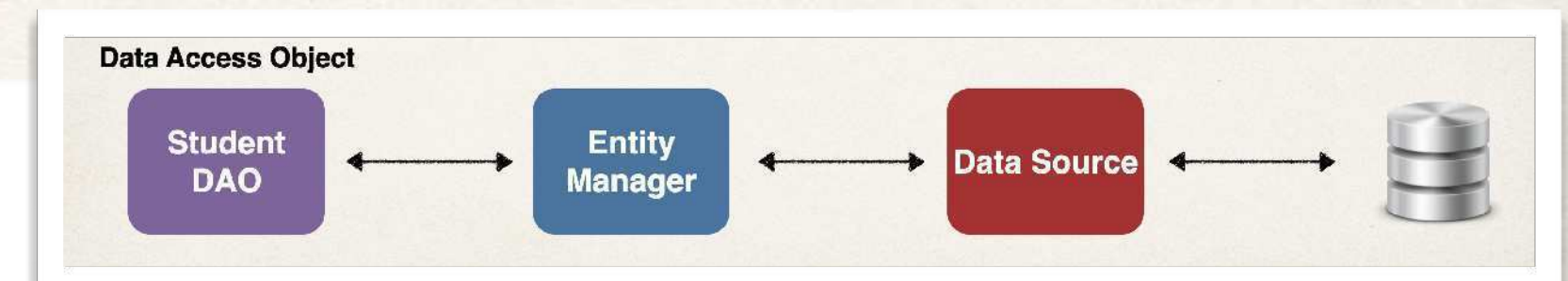
    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

Inject the Entity Manager

Save the
Java object



Spring @Transactional

- Spring provides an **@Transactional** annotation
- **Automagically** begin and end a transaction for your JPA code
 - No need for you to explicitly do this in your code
- This Spring **magic** happens behind the scenes

Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;

public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;

    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

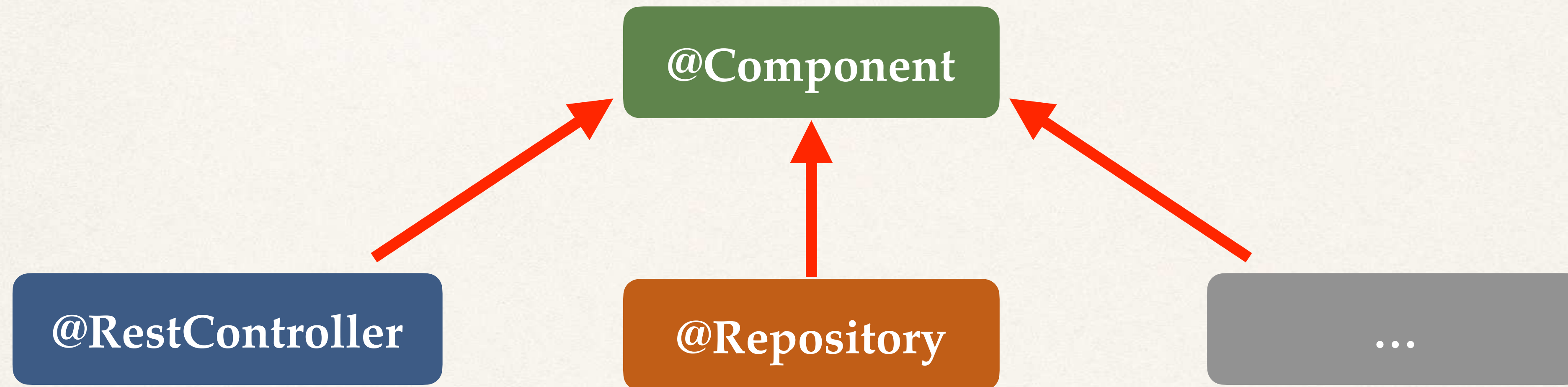
    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```

Handles transaction
management

Specialized Annotation for DAOs

- Spring provides the **@Repository** annotation



Specialized Annotation for DAOs

- Applied to DAO implementations
- Spring will automatically register the DAO implementation
 - thanks to component-scanning
- Spring also provides translation of any JDBC related exceptions

Step 2: Define DAO implementation

Specialized annotation
for repositories

Supports component
scanning

Translates JDBC
exceptions

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class StudentDAOImpl implements StudentDAO {

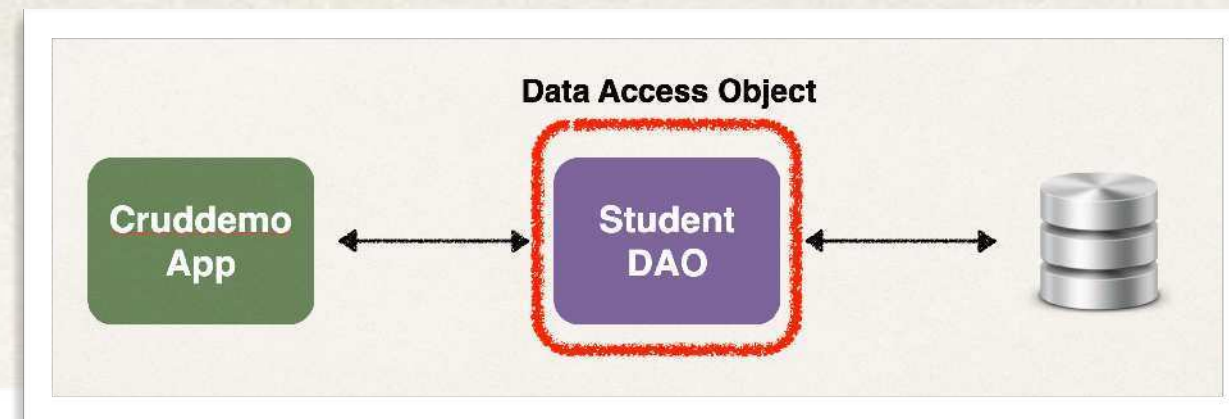
    private EntityManager entityManager;

    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    @Transactional
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }

}
```


Step 3: Update main app



Inject the StudentDAO

```
@SpringBootApplication
public class CruddemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(CruddemoApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
        return runner -> {

            createStudent(studentDAO);

        }

        private void createStudent(StudentDAO studentDAO) {

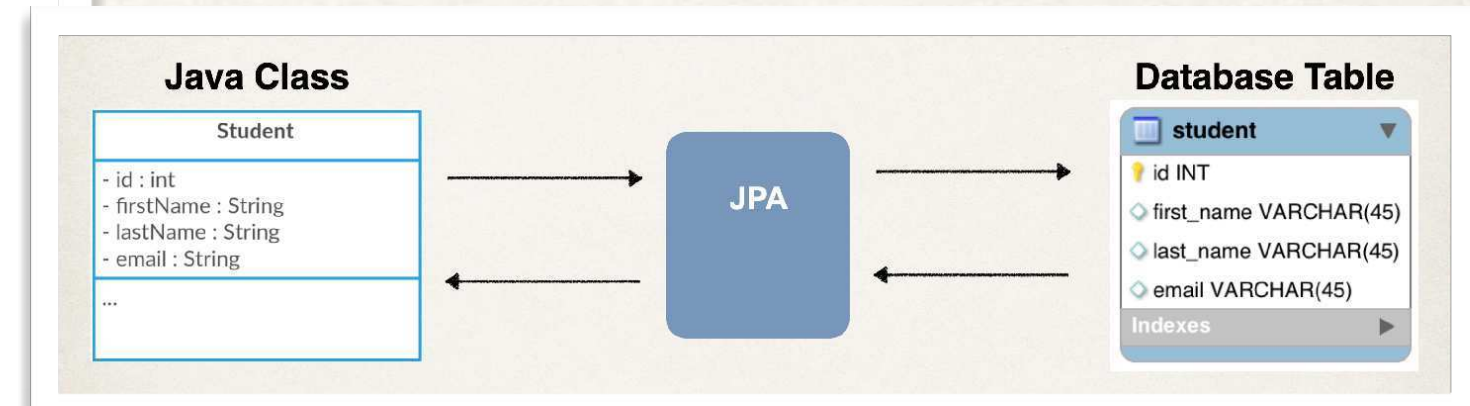
            // create the student object
            System.out.println("Creating new student object...");
            Student tempStudent = new Student("Paul", "Doe", "paul@luv2code.com");

            // save the student object
            System.out.println("Saving the student...");
            studentDAO.save(tempStudent);

            // display id of the saved student
            System.out.println("Saved student. Generated id: " + tempStudent.getId());

        }

    }
}
```



Retrieving an Object



JPA CRUD Apps

- Create objects

→ Read objects

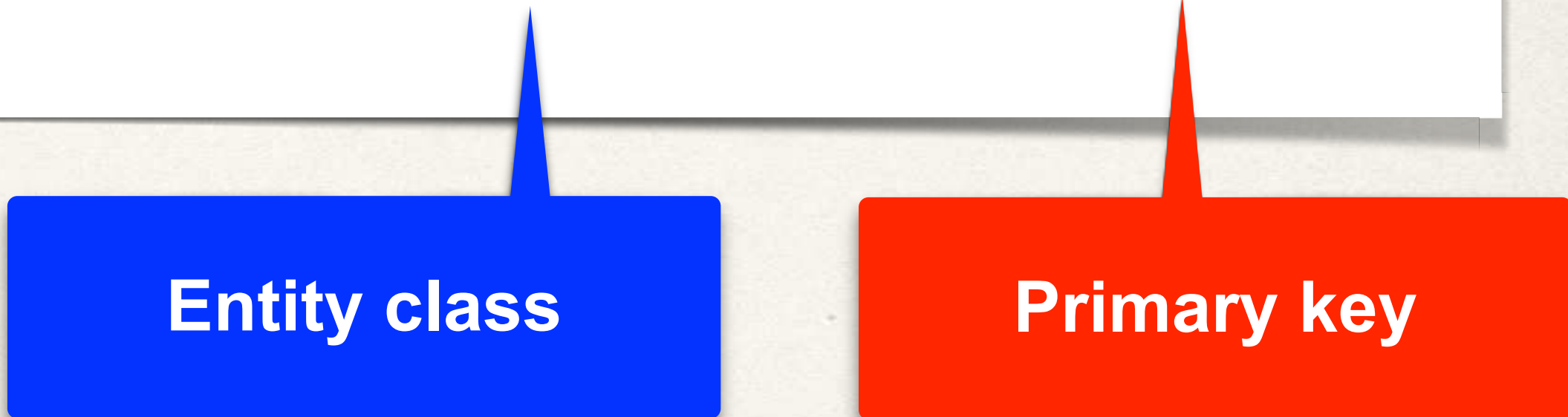
- Update objects

- Delete objects

Retrieving a Java Object with JPA

*// retrieve/read from database using the **primary key**
// in this example, retrieve Student with **primary key: 1***

```
Student myStudent = entityManager.find(Student.class, 1);
```



Entity class

Primary key

Development Process

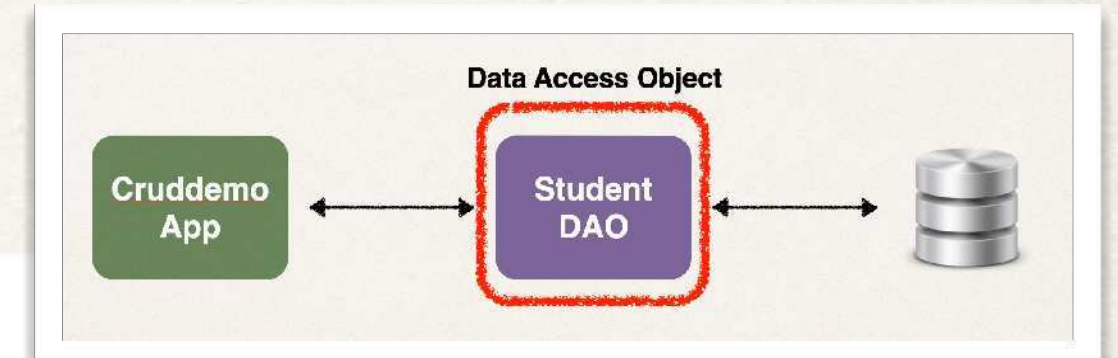
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app



Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    ...
    Student findById(Integer id);
}
```



Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
...

public class StudentDAOImpl implements StudentDAO {
    private EntityManager entityManager;
    ...

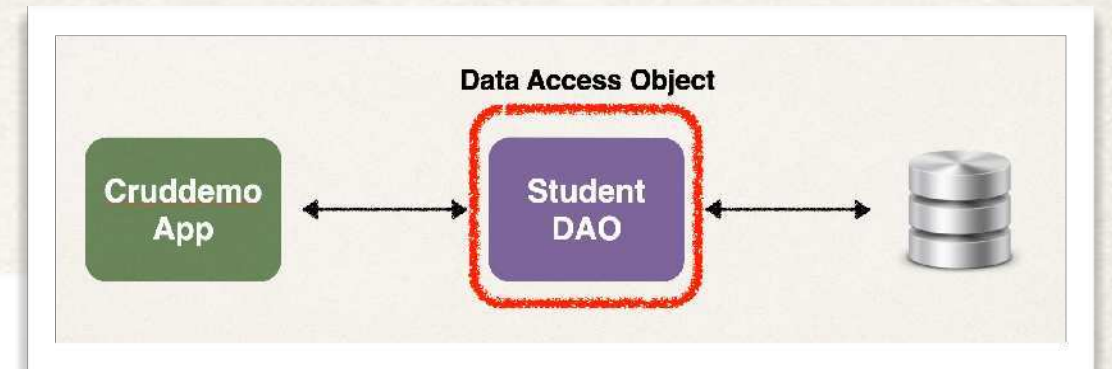
    @Override
    public Student findById(Integer id) {
        return entityManager.find(Student.class, id);
    }
}
```

No need to add `@Transactional` since we are doing a query

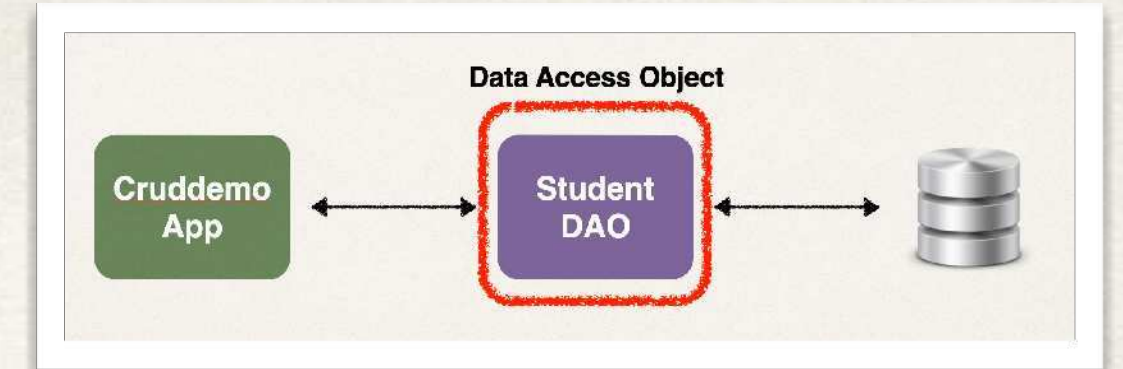
If not found, returns null

Entity class

Primary key



Step 3: Update main app



```
@SpringBootApplication
public class CruddemoApplication {
    ...

    @Bean
    public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
        return runner -> {
            readStudent(studentDAO);
        };
    }
    ...
}
```

```
private void readStudent(StudentDAO studentDAO) {
    // create a student object
    System.out.println("Creating new student object...");
    Student tempStudent = new Student("Daffy", "Duck", "daffy@luv2code.com");

    // save the student object
    System.out.println("Saving the student...");
    studentDAO.save(tempStudent);

    // display id of the saved student
    System.out.println("Saved student. Generated id: " + tempStudent.getId());

    // retrieve student based on the id: primary key
    System.out.println("\nRetrieving student with id: " + tempStudent.getId());

    Student myStudent = studentDAO.findById(tempStudent.getId());

    System.out.println("Found the student: " + myStudent);
}
```


Query Objects



JPA CRUD Apps

- Create objects

- Read objects

- Update objects

- Delete objects

JPA Query Language (JPQL)

- Query language for retrieving objects
- Similar in concept to SQL
 - where, like, order by, join, in, etc...
- However, JPQL is based on **entity name** and **entity fields**

Retrieving all Students

Java Class

Student
- id : int - firstName : String - lastName : String - email : String ...

Name of JPA Entity ...
the class name

```
TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);  
List<Student> students = theQuery.getResultList();
```

Note: this is NOT the name of the database table

**All JPQL syntax is based on
entity name and entity fields**

Retrieving Students: `lastName = 'Doe'`

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
    "FROM Student WHERE lastName='Doe'", Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

Retrieving Students using OR predicate:

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
    "FROM Student WHERE lastName='Doe' OR firstName='Daffy'", Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Field of JPA Entity

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

Retrieving Students using LIKE predicate:

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
    "FROM Student WHERE email LIKE '%luv2code.com'", Student.class);  
  
List<Student> students = theQuery.getResultList();
```

Match of email addresses
that ends with
luv2code.com

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

JPQL - Named Parameters

```
public List<Student> findByLastName(String theLastName) {  
  
    TypedQuery<Student> theQuery = entityManager.createQuery(  
                                                "FROM Student WHERE lastName=:theData", Student.class);  
  
    theQuery.setParameter("theData", theLastName);  
  
    return theQuery.getResultList();  
}
```



JPQL Named Parameters are
prefixed with a colon :

Think of this as a placeholder
that is filled in later

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

JPQL - select clause

- The query examples so far did not specify a “select” clause
- The Hibernate implementation is lenient and allows Hibernate Query Language (HQL)
- For strict JPQL, the “select” clause is required

```
TypedQuery<Student> theQuery =  
    entityManager.createQuery("select s FROM Student s", Student.class);
```

s is an “identification variable” / alias
Provides a reference to the Student entity object

s - Can be any name
Useful for when you have complex queries

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

JPQL - select clause

- Other examples, for strict JPQL

Java Class

Student

- id : int
- firstName : String
- lastName : String
- email : String

...

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
    "select s FROM Student s WHERE s.email LIKE '%luv2code.com'", Student.class);
```

```
TypedQuery<Student> theQuery = entityManager.createQuery(  
    "select s FROM Student s WHERE s.lastName=:theData", Student.class);
```


Development Process



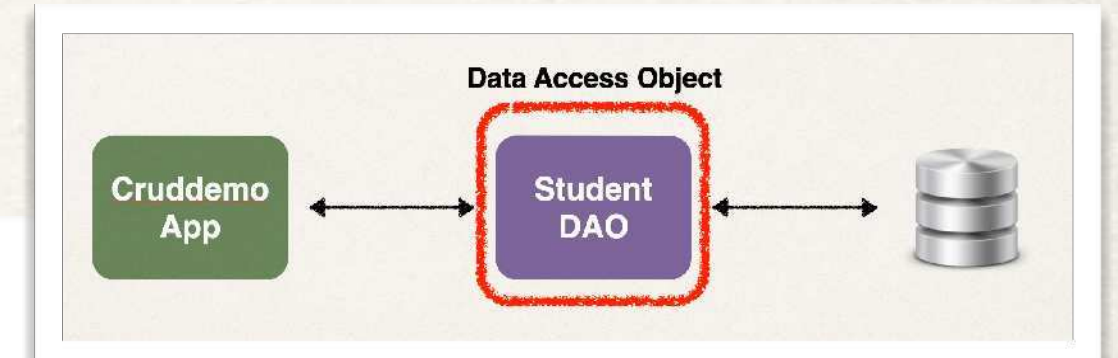
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step 1: Add new method to DAO interface

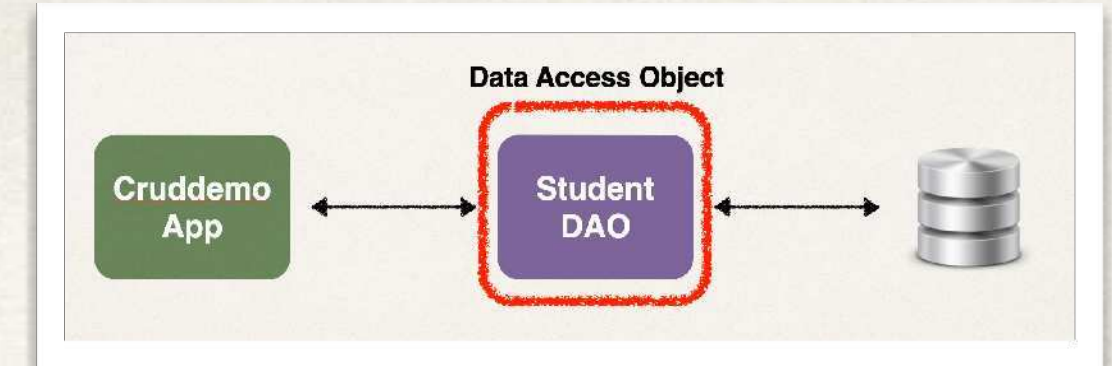
```
import com.luv2code.cruddemo.entity.Student;
import java.util.List;

public interface StudentDAO {
    ...

    → List<Student> findAll();
}
```



Step 2: Define DAO implementation



```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import jakarta.persistence.TypedQuery;
import java.util.List;
...

public class StudentDAOImpl implements StudentDAO {

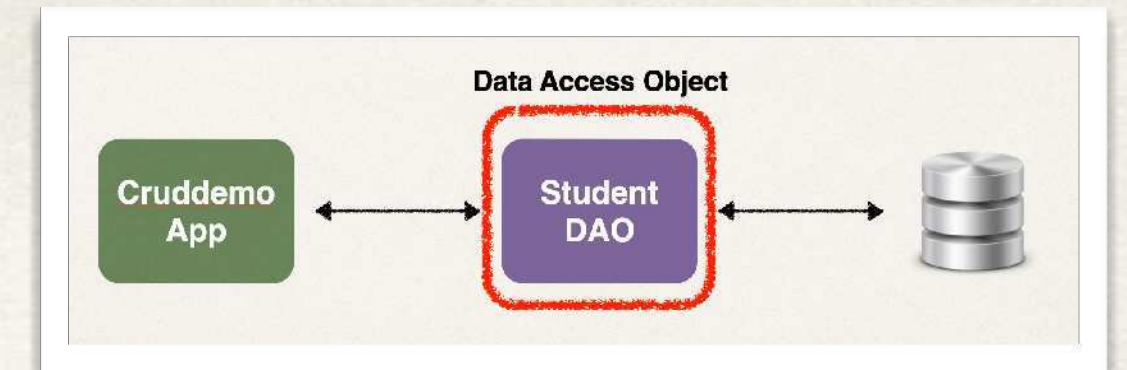
    private EntityManager entityManager;
    ...

    @Override
    public List<Student> findAll() {
        TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);
        return theQuery.getResultList();
    }
}
```

No need to add `@Transactional` since we are doing a query

Name of JPA Entity

Step 3: Update main app



```
@SpringBootApplication
public class CruddemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(CruddemoApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
        return runner -> {
            queryForStudents(studentDAO);
        };
    }

    private void queryForStudents(StudentDAO studentDAO) {

        // get list of students
        List<Student> theStudents = studentDAO.findAll();

        // display list of students
        for (Student tempStudent : theStudents) {
            System.out.println(tempStudent);
        }
    }
}
```


Updating an Object



JPA CRUD Apps

- Create objects

- Read objects

- Update objects

- Delete objects

Update a Student

```
Student theStudent = entityManager.find(Student.class, 1);
```

```
// change first name to "Scooby"  
theStudent.setFirstName("Scooby");
```

```
entityManager.merge(theStudent);
```



Update the entity

Update last name for all students

```
int numRowsUpdated = entityManager.createQuery(  
    "UPDATE Student SET lastName='Tester'")  
    .executeUpdate();
```

Field of JPA Entity

Return the number
of rows updated

Execute this
statement

Name of JPA Entity ...
the class name

Java Class

Student

- id : int
- firstName : String
- lastName : String
- email : String

...

Development Process

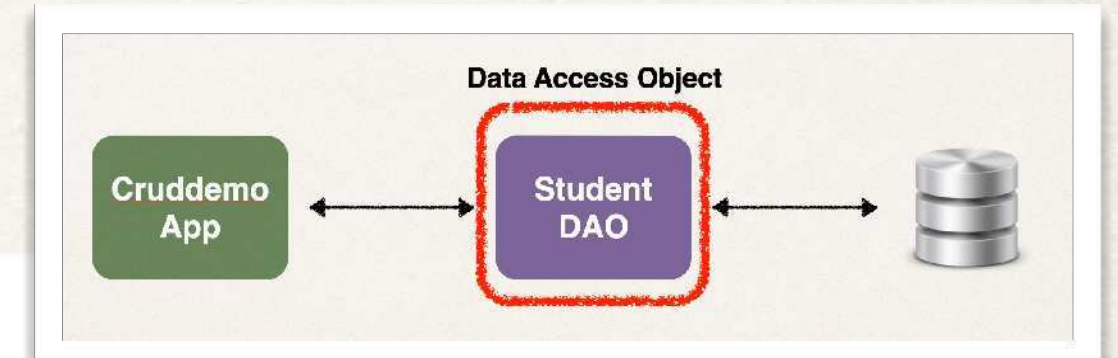
1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step-By-Step

Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    ...
    void update(Student theStudent);
}
```



Step 2: Define DAO implementation

```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.transaction.annotation.Transactional;
...

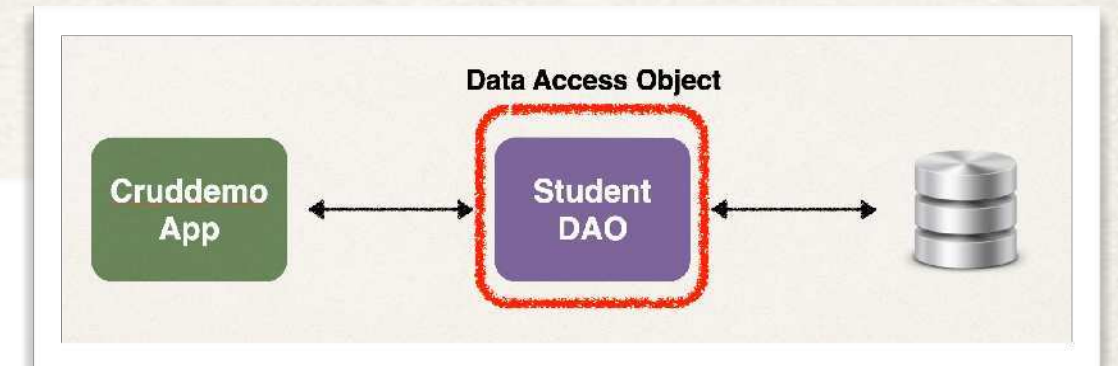
public class StudentDAOImpl implements StudentDAO {

    private EntityManager entityManager;
    ...

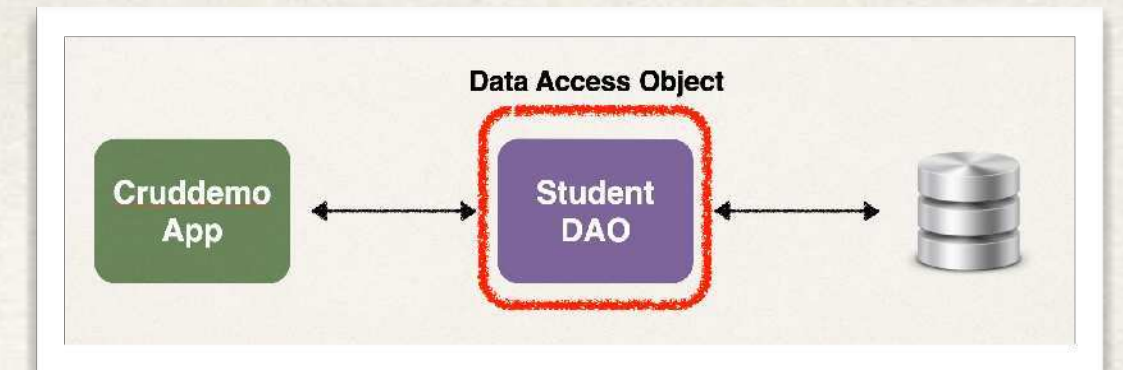
    @Override
    @Transactional
    public void update(Student theStudent) {
        entityManager.merge(theStudent);
    }

}
```

Add @Transactional since we are performing an update



Step 3: Update main app



```
@SpringBootApplication
public class CruddemoApplication {
    ...

    @Bean
    public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
        return runner -> {
            ➡ updateStudent(studentDAO);
        };
    }
    ...
}
```

```
private void updateStudent(StudentDAO studentDAO) {

    // retrieve student based on the id: primary key
    int studentId = 1;
    System.out.println("Getting student with id: " + studentId);

    Student myStudent = studentDAO.findById(studentId);

    System.out.println("Updating student...");

    // change first name to "Scooby"
    myStudent.setFirstName("Scooby");
    studentDAO.update(myStudent);

    // display updated student
    System.out.println("Updated student: " + myStudent);
}
```


Deleting an Object



JPA CRUD Apps

- Create objects
- Read objects
- Update objects
- Delete objects

Delete a Student

```
// retrieve the student  
int id = 1;  
Student theStudent = entityManager.find(Student.class, id);  
  
// delete the student  
entityManager.remove(theStudent);
```


Delete based on a condition

Field of JPA Entity

```
int numRowsDeleted = entityManager.createQuery(  
    "DELETE FROM Student WHERE lastName='Smith'")  
    .executeUpdate();
```

Return the number of
rows deleted

Execute this
statement

Name of JPA Entity ...
the class name

Method name "Update" is a generic term
We are "modifying" the database

Java Class

Student

- id : int
- firstName : String
- lastName : String
- email : String

...

Delete All Students

```
int numRowsDeleted = entityManager
                        .createQuery( "DELETE FROM Student" )
                        .executeUpdate( ) ;
```

Java Class

Student
- id : int - firstName : String - lastName : String - email : String
...

Development Process

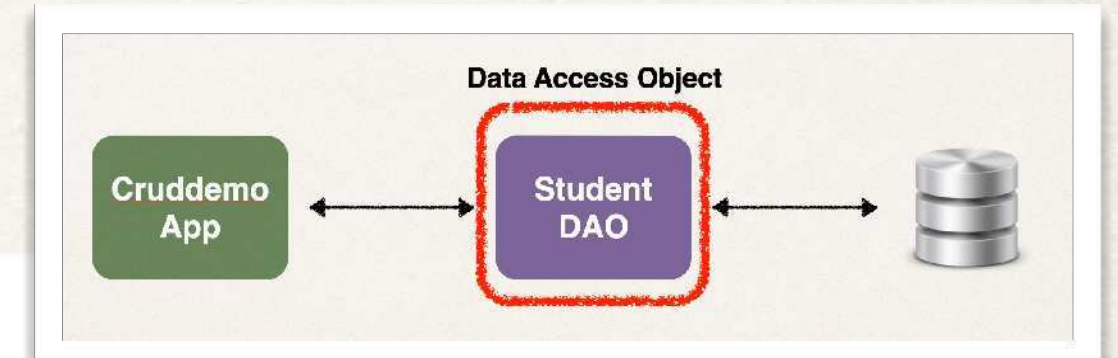


1. Add new method to DAO interface
2. Add new method to DAO implementation
3. Update main app

Step 1: Add new method to DAO interface

```
import com.luv2code.cruddemo.entity.Student;

public interface StudentDAO {
    ...
    void delete(Integer id);
}
```



Step 2: Define DAO implementation

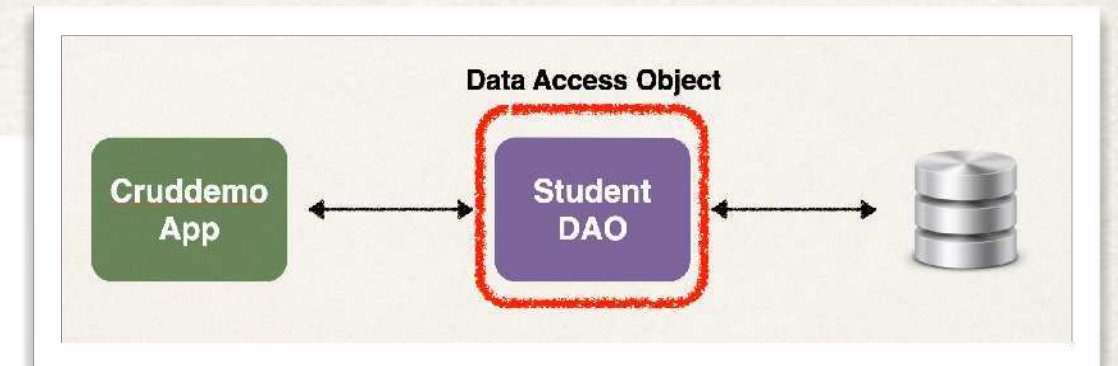
```
import com.luv2code.cruddemo.entity.Student;
import jakarta.persistence.EntityManager;
import org.springframework.transaction.annotation.Transactional;
...

public class StudentDAOImpl implements StudentDAO {

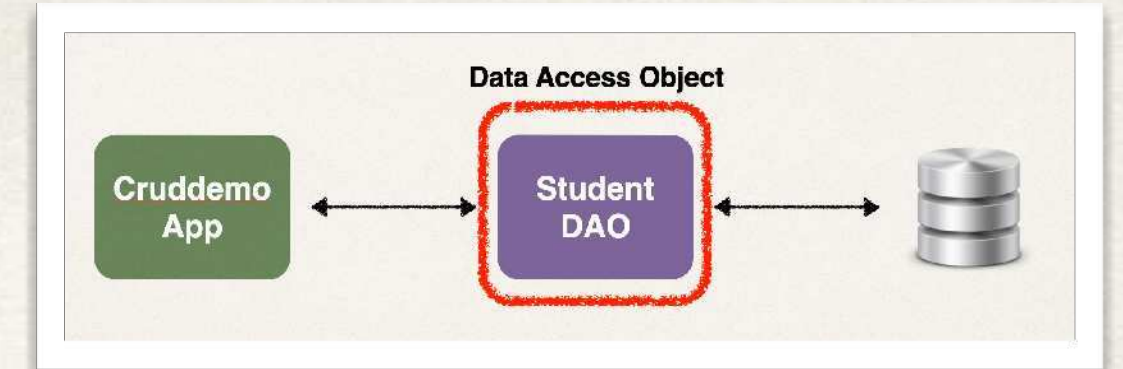
    private EntityManager entityManager;
    ...

    @Override
    @Transactional
    public void delete(Integer id) {
        Student theStudent = entityManager.find(Student.class, id);
        entityManager.remove(theStudent);
    }
}
```

Add @Transactional since we are performing a delete



Step 3: Update main app



```
@SpringBootApplication
public class CruddemoApplication {
    ...

    @Bean
    public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
        return runner -> {
            deleteStudent(studentDAO);
        };
    }
    ...
}
```

```
private void deleteStudent(StudentDAO studentDAO) {
    // delete the student
    int studentId = 3;

    System.out.println("Deleting student id: " + studentId);

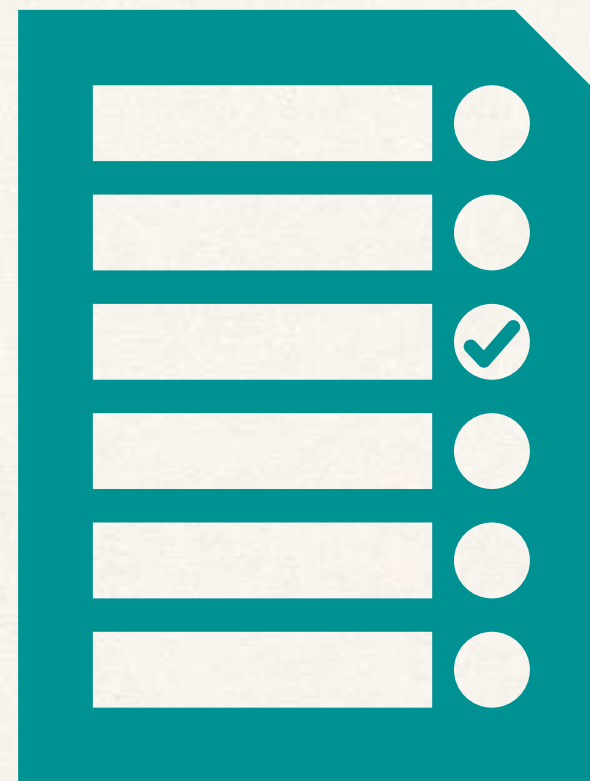
    studentDAO.delete(studentId);
}
```


Create Database Tables from Java Code



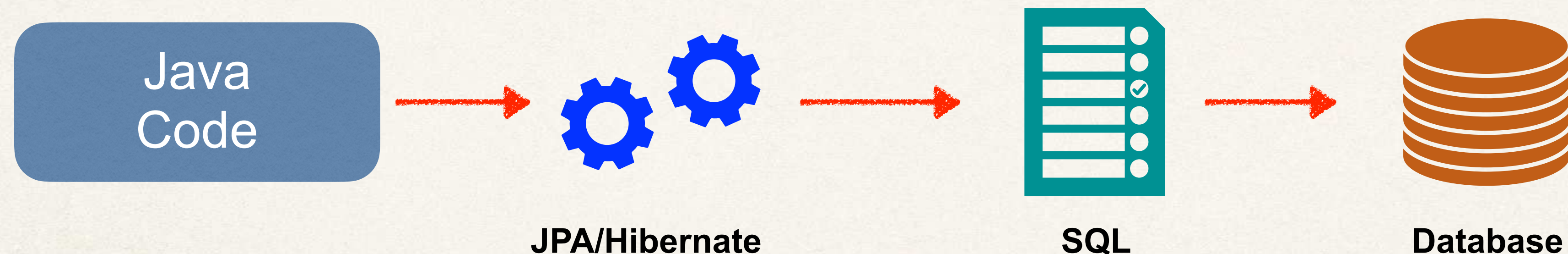
Create database tables: **student**

- Previously, we created database tables by running a SQL script



Create database tables: student

- JPA / Hibernate provides an option to automagically create database tables
- Creates tables based on Java code with JPA / Hibernate annotations
- Useful for development and testing



Configuration

- In Spring Boot configuration file: **application.properties**

```
spring.jpa.hibernate.ddl-auto=create
```

- When you run your app, JPA / Hibernate will **drop** tables then **create** them
- Based on the JPA / Hibernate annotations in your Java code

Creating Tables based on Java Code

Hibernate will generate and execute this

2

create table student (id integer not null auto_increment,
email varchar(255), first_name varchar(255),
last_name varchar(255), primary key (id))

1

```
@Entity
@Table(name="student")
public class Student {

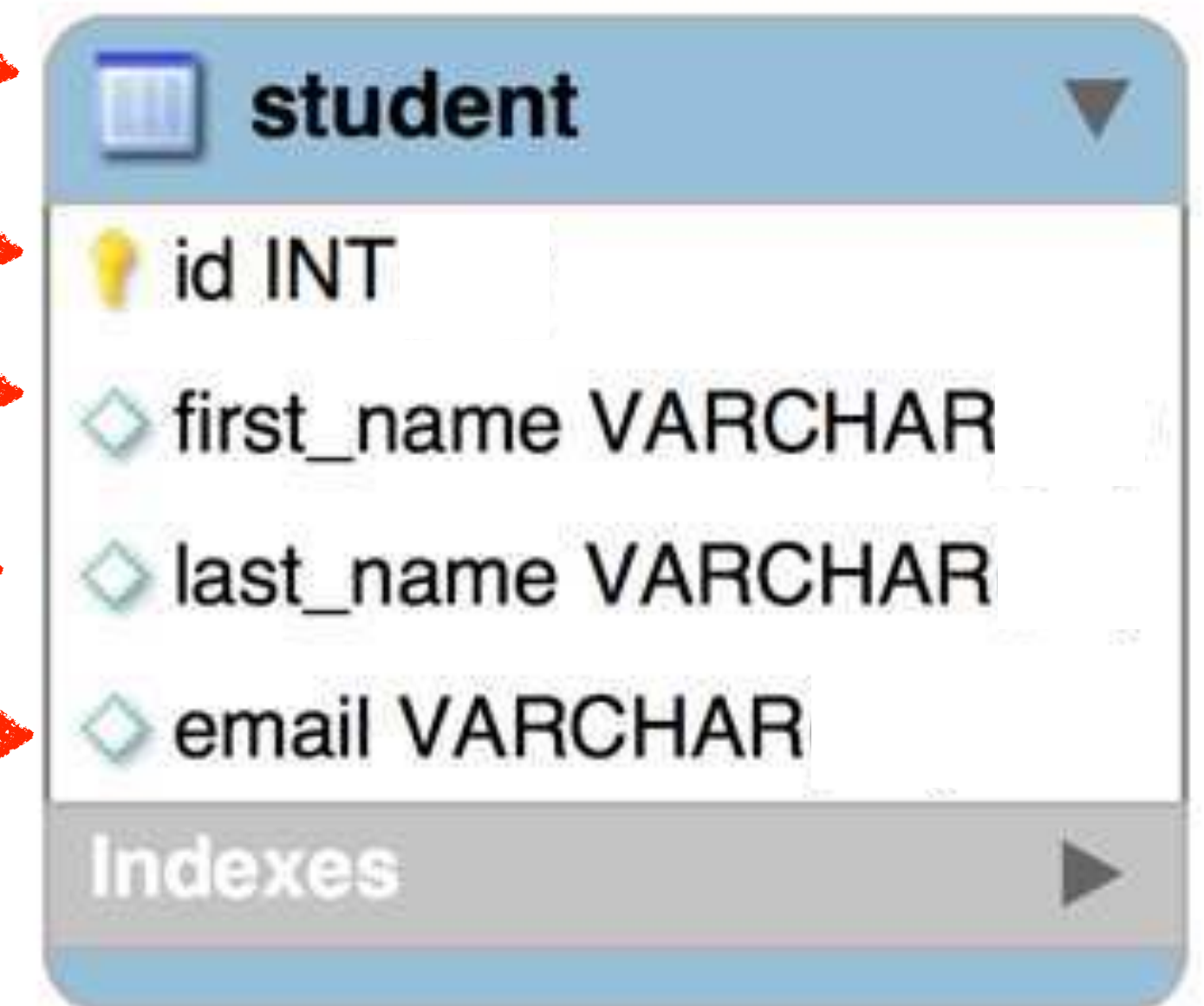
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email")
    private String email;

    ...
    // constructors, getters / setters
}
```



Configuration - application.properties

```
spring.jpa.hibernate.ddl-auto=PROPERTY-VALUE
```

Property Value	Property Description
none	No action will be performed
create	Database tables are dropped followed by database tables creation
create-drop	Database tables are dropped followed by database tables creation. On application shutdown, drop the database tables
validate	Validate the database tables schema
update	Update the database tables schema

When database tables are dropped, all data is lost

Basic Projects

- For basic projects, can use auto configuration

```
spring.jpa.hibernate.ddl-auto=create
```

- Database tables are dropped first and then created from scratch

Note:

When database tables are dropped, all data is lost

Basic Projects

- If you want to create tables once ... and then keep data, use: update

```
spring.jpa.hibernate.ddl-auto=update
```

- However, will ALTER database schema based on latest code updates
- Be VERY careful here ... only use for basic projects



Warning

```
spring.jpa.hibernate.ddl-auto=create
```

- Don't do this on **Production** databases!!!
- You don't want to drop your Production data
 - **All data is deleted!!!**
- Instead for Production, you should have DBAs run SQL scripts



Use Case

```
spring.jpa.hibernate.ddl-auto=create
```

- Automatic table generation is useful for
 - Database integration testing with in-memory databases
 - Basic, small hobby projects

Recommendation

- In general, I don't recommend auto generation for enterprise, real-time projects
- You can VERY easily drop PRODUCTION data if you are not careful 
- I recommend SQL scripts 
 - Corporate DBAs prefer SQL scripts for governance and code review
 - The SQL scripts can be customized and fine-tuned for complex database designs
 - The SQL scripts can be version-controlled
 - Can also work with schema migration tools such as Liquibase and Flyway